A Draft Proposal to define an Extensible Runtime Containment and Services Protocol for JavaBeans (Version 0.97)

Laurence Cable and Graham Hamilton.

THIS IS A DRAFT SPECIFICATION, IT IS THEREFORE SUBJECT TO CHANGE, AND FURTHERMORE IMPLIES NO INTENT ON BEHALF OF JavaSoft TO DELIVER SUCH BEHAVIOR

Send comments to java-beans@java.sun.com.

1.0 Introduction.

Currently the JavaBeans specification (Version 1.0) contains neither conventions describing a hierarchy or logical structure of JavaBeans, nor conventions for those JavaBeans to rendezvous with, or obtain arbitrary services or facilities from, the execution environment within which the JavaBean was instantiated.

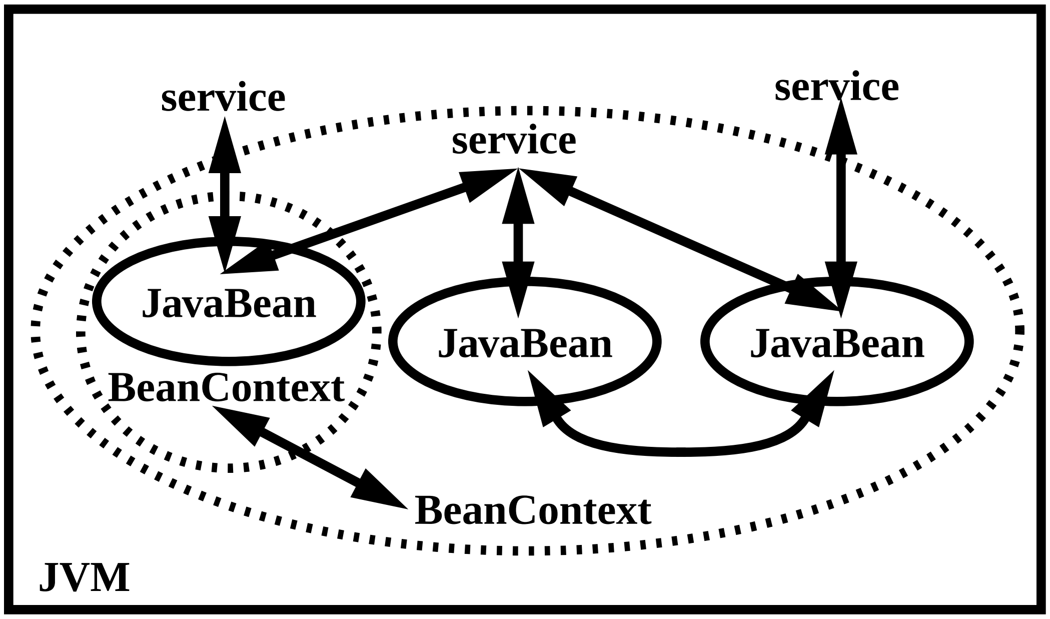
It is desirable to both provide a logical, traversable, hierarchy of JavaBeans, and further to provide a general mechanism whereby an object instantiating an arbitrary JavaBean can offer that JavaBean a variety of services, or interpose itself between the underlying system service and the JavaBean, in a conventional fashion.

In other component models there exists the concept of a relationship between a Component and its environment, or Container, wherein a newly instantiated Component is provided with a reference to its Container or Embedding Context.

The Container, or Embedding Context not only establishes the hierarchy or logical structure, but its also acts as a service provider that Components may interrogate in order to determine, and subsequently employ, the services provided by their Context.

This proposal defines such a protocol that supports extensible mechanisms that:

- Introduce an abstraction for the environment, or context, in which a JavaBean logically functions during its lifecycle, that is a hierarchy or structure of JavaBeans.
- Enable the dynamic addition of arbitrary services to a JavaBean’s environment.
- Provide a single service discovery mechanism through which JavaBeans may interrogate their environment in order both to ascertain the availability of particular services and to subsequently employ those services.
- Provide a simple mechanism to propagate an Environment to a JavaBean.
- Provide better support for JavaBeans that are also Applets.



2.0 API Proposal

2.1 interface java.beans.BeanContext

Since the primary roles of a JavaBean's environment, or *BeanContext*[1], is to:

1. Provide a hierarchy, or logical structure for nested JavaBeans and *BeanContext*s.
2. Provide a mechanism for rendezvous between a JavaBean, and a variety of services and information available from the rest of the execution environment.

This *BeanContext* may be best modeled by an interface that defines the structure or hierarchy primitives, including an Aggregation interface, as proposed elsewhere, to provide for a generic service discovery and provider facility, therefore:

```
public interfacejava.beans.BeanContext
        extends java.beans.BeanContextChild,
                java.util.Collection {

     Object instantiateChild(String beanName)
```

1. Existing Component architectures use the term "Container" to refer to the entity that provides a "Component" with services etc. from the execution environment, and "Containment" as the relationship between the "Container" and "Component". These terms are greatly overloaded in the industry, and in particular are already used in the context of Java. Additionally the intended usage of this facility is far broader than is implied by the general usage of the term "Container", therefore this proposal uses a new term "BeanContext" to describe the "Container" entity.

```
            throws IOException, ClassNotFoundException;

     Object  getService(Class            serviceClass,
                        BeanContextChild requestor
             );
     boolean hasService(Class            serviceClass,
                        BeanContextChild requestor
             );

     public InputStream
          getResourceAsStream(String           name,
                              BeanContextChild requestor
          );

     public java.net.URL
          getResource(String           name,
                      BeanContextChild requestor
          );

     void addPropertyChangeListener
               (String name,PropertyChangeListener pcl);

     void removePropertyChangeListener
               (String name, PropertyChangeListener pcl);

     void removeBeanContextListener
               (BeanContextListener bcl);

     void addBeanContextListener(BeanContextListener bcl);
}
```

2.1.1 java.beans.BeanContextListener & BeanContextEvent

```
public interface BeanContextListener
       extends   java.util.EventListener {
     void beanContextChanged(BeanContextEvent bce);
}
```

```
public abstract class BeanContextEvent
       extends  java.util.EventObject {
     public BeanContext getBeanContext();

     public synchronized void
          setPropagatedFrom(BeanContext bc);

     public synchronized BeanContext getPropagatedFrom();

     public synchronized boolean isPropagated()
}

public abstract class BeanContextMembershipEvent
       extends BeanContextEvent {
     public boolean isDeltaMember(Object o);

     public Object[] getDeltas();

     public boolean isChildrenAddedEvent();

     public boolean isChildrenRemovedEvent()
}

public BeanContextAddedEvent
       extends BeanContextMembershipEvent {
     BeanContextAddedEvent(BeanContext bc, Object[] bccs);
}
public BeanContextRemovedEvent
       extends BeanContextMembershipEvent {
     BeanContextRemovedEvent(BeanContext bc, Object[] bccs);
}
```

The *java.beans.BeanContextListener* interface is intended to provide a mechanism to allow entities in the system to monitor changes in a particular context instance. As detailed in the following section, a *beanContextChanged*() method notification is fired whenever a state change occurs in a particular *BeanContext* instance that the *BeanContext* implementation wishes to expose to its Listeners. The associated *BeanContextEvent* instance describes the nature of the change.

Instances of *java.beans.BeanContextListener* are registered and unregistered with a particular *BeanContext* instance via its *addBeanContextListener()* and *removeBeanContextListener()* methods.

Note that the *BeanContextEvent* provides a mechanism whereby an entity receiving such an event can determine if it has been propagated from a *BeanContext* nested with the BeanContext upon which the entity registered its associated *BeanContextListener* interface, via the *isPropagatedFrom*() and *getPropagatedFrom*() methods.

Note that a *BeanContext* is not required to propagate *BeanContextListener* notifications it receives to its *BeanContextListeners*, since there are performance implications in doing so, however the protocol is provided for those applications that require knowledge of membership changes throughout the hierarchy.

The *BeanContextMembershipEvent* describes changes that occur in the membership of a particular *BeanContext* instance. This event encapsulates the list of children either added to, or removed from, the membership of a particular *BeanContext* instance, i.e the delta in the membership set.

whenever a successful *add*(), *remove*(), *addAll*(), or *clear*() is invoked upon a particular BeanContext instance, a *BeanContextMembershipEvent* is fired describing the children effected by the operation.

2.1.2 The BeanContext as a participant in nested structure

One of the roles of the *BeanContext* is to introduce the notion of a hierarchical nesting or structure of *BeanContext* and JavaBean instances. In order to model this structure the *BeanContext* must expose API that define the relationships in the structure or hierarchy.

The *BeanContext* exposes its superstructure through implementation of the *java.beans.BeanContextChild* interface. This interface allows the discovery and manipulation of a *BeanContext*'s nesting *BeanContext*, and thus introduces a facility to create a hierarchy of *BeanContexts*.

The *BeanContext* exposes its substructure through a number of interface methods modelled by the *java.util.Collection* interface:

The *add()* method may be invoked in order to nest a new JavaBean or *BeanContext* within the target *BeanContext*. A conformant *add()* implementation is required to adhere to the following semantics:

- Each child object shall appear only once in the set of children for a given *BeanContext*. If the instance is already a member of the *BeanContext* then the method shall throw *IllegalArgumentException*.
- Each valid child shall be added to the set of children of a given source *BeanContext,* and thus shall appear in the set of children, obtained through either the *toArray(),* or *iterator()* methods, until such time as that child is deleted from the nesting *BeanContext* via an invocation of *remove()*.
- As the child is added to the set of nested children, and where that child implements the *java.beans.beancontext.BeanContextChild* interface, the *BeanContext* shall invoke the *setBeanContext*() method upon that child, with a reference to itself. Upon invocation, a

child may, if it is for some reason unable or unprepared to function in that *BeanContext*, throw a *PropertyVetoException* to notify the nesting *BeanContext*. If the child throws such an exception the *BeanContext* shall revoke the addition of the child to the set of nested children and throw an *IllegalArgumentException*.

- Once the *targetChild* has been successfully processed, the *BeanContext* shall fire a *java.beans.beancontext.BeanContextAddedEvent*, containing a reference to the newly added *targetChild*, to the Listeners currently registered to receive *BeanContextListener* notifications.
- JavaBeans that implement the *java.beans.Visibility* interface shall be notified via the appropriate method, either *dontUseGui()* or *okToUseGui(),* of their current ability to render GUI as defined the policy of the *BeanContext*.
- If the newly added child implements *BeanContextChild,* the *BeanContext* shall register itself with the child on both its *VetoableChangeListener* and *PropertyChangeListener* interfaces to monitor, at least, that *BeanContextChild*'s "beanContext" property.

  By doing so the *BeanContext* can monitor its child and can detect when such children are removed from their Context by a 3rd party invoking *setBeanContext*(). A *BeanContext* may veto such a change by a 3rd party if it determines that the child is not in a state to depart membership of that Context at that time.
- If the JavaBean(s) added, implement Listener interfaces that the *BeanContext* is a source for, then the *BeanContext* may register the newly added objects via the appropriate Listener registration methods as a permissable side effect of nesting.
- The method shall return `true` when complete.

The *remove()* method may be invoked in order to remove an existing child JavaBean or *BeanContext* from within the target *BeanContext*. A conformant *remove()* implementation is required to adhere to the following semantics:

- If a particular child is not present in the set of children for the source *BeanContext,* the method shall throw *IllegalArgumentException*.
- Remove the valid *targetChild* from the set of children for the source *BeanContext,* also removing that child from any other Listener interfaces that it was implicitly registered on, for that *BeanContext*.

  Subsequently, if the *targetChild* implements the *java.beans.beancontext.BeanContextChild* interface, the *BeanContext* shall invoke the setBeanContext() with a `null`[1] *BeanContext* value, in order to notify that child that it is no longer nested within the *BeanContext*.

1. Note, if the *remove*() was invoked as a result of the *BeanContext* receiving an unexpected *PropertyChangeEvent* notification as a result of a 3rd party invoking *setBeanContext*() then the remove implementation shall not invoke *setBeanContext*(`null`) on that child as part of the *remove*() semantics, since to do so would overwrite the value previously set by the 3rd party..

If a particular *BeanContextChild* is in a state where it is not able to be unnested from its nesting *BeanContext* it may throw a *PropertyVetoException*, upon receipt of this the *BeanContext* shall revoke the remove operation for this instance. To avoid infinite recursion children are not permitted to veto subsequent remove notifications.

Once the *targetChild* has been removed from the set of children, the *BeanContext* shall fire a *java.beans.BeanContextRemovedEvent,* containing a reference to the *targetChild* just removed, to the Listeners currently registered to receive *BeanContextListener* notifications.

- *If the targetChild implements* java.beans.BeanContextChild then the *BeanContext* shall deregister itself from that child's *PropertyChangeListener* and *VetoableChangeListener* sources.
- If the *BeanContext* had previously registered the object(s) removed, as Listeners on events sources implemented by the *BeanContext*, as a side effect of nesting those objects, then the *BeanContext* shall de-register the newly removed object from the applicable source(s) via the appropriate Listener de-registration method(s)
- Finally the method shall return the value `true`.

Note that the lifetime of any child of a nesting *BeanContext*, is at least for the duration of that child's containment within a given *BeanContext*. For simple JavaBeans that are not aware of their containment within a *BeanContext*, this usually implies that the JavaBean shall exist for the lifetime of the nesting *BeanContext*.

The *toArray(),* method shall return a copy of the current set of JavaBean or *BeanContext* instances nested within the target *BeanContext*, and the *iterator*() method shall supply a *java.util.Iterator* object over the same set of children.

The *contains()* method shall return `true` if the object specified is currently a child of the *BeanContext*.

The *size()* method returns the current number of children nested.

The *isEmpty*() method returns true iff the *BeanContext* has no children.

BeanContext's are not required to implement either *addAll(Collection c)* or *clear()* methods defined by *java.util.Collection*, however if they do they must implement the semantics defined, per object, for both *add*() and *remove*(). In the failure cases these methods should revoke any partially applied changes to return the *BeanContext* to the state it was in prior to the composite operation being invoked, no BeanContextEvents shall be fired in the failure case..

Note that *all the Collection* methods all require proper synchronization in order to function correctly in a multi-threaded environment.

The *instantiateChild()* method is a convenience method that may be invoked to instantiate a new JavaBean instance as a child of the target *BeanContext*. The implementation of the

JavaBean is derived from the value of the *beanName* actual parameter, and is defined by the *java.beans.Beans.instantiate()* method.

Typically, this shall be implemented by calling the appropriate *java.beans.Beans.instantiate()* method, using the *ClassLoader* of the target *BeanContext*.

2.1.3 Resources.

The *BeanContext* defines two methods; *getResourceAsStream*() and *getResource*() which are analogous to those methods found on *java.lang.ClassLoader. BeanContextChild* instances nested within a *BeanContext* shall invoke the methods on their nesting Context in preference for those on *ClassLoader,* to allow a *BeanContext* implementation to augment the semantics by interposing behavior between the child and the underlying *ClassLoader* semantics

2.1.4 The BeanContext as a Service Provider

Using the *hasService()* and *getService()* methods of the *BeanContext*, JavaBeans can interrogate for the existence of, and subsequently obtain references to, a variety of dynamic services from its environment, See "Standard/Suggested Conventions for BeanContext Delegates" on page 13.

In the case when a nested *BeanContext* is requested for a particular Delegate that it has no implementation for, then the *BeanContext* may delegate the delegation requested to its own nesting *BeanContext* in order to be satisfied. Thus Delegation requests can propagate from the leaf JavaBeans to the root *BeanContext.* This is strongly recommended since it has a significant impact upon interoperability.

Using this mechanism to dynamically discover and utilize services, decouples JavaBeans and *BeanContext*s, enabling both greater independence of JavaBeans from their environment and significant improvements in portability.

The set of Delegate types of a *BeanContext* is variable over the lifetime of the *BeanContext.*

For any arbitrary Delegate, it is valid a valid reference at least until the last child referencing it maintains that reference.

When *BeanContextChild* instances are removed from a particular *BeanContext* instance, they shall discard all references to any Delegates they obtained from that *BeanContext*. Futhermore, *BeanContexts* are not permitted to expose Delegates, obtained by defering the request to their nesting *BeanContext,* to their children where the nature of the Delegate obtained is such that its function depends upon some aspect of the nesting relationship between the initial referring BeanContext, and the nesting ancestor that satisfies that Delegation.

2.1.5 The role of a BeanContext in Persistence

Since one of the primary roles of a *BeanContext* is to represent a logical nested structure of JavaBean and *BeanContext* instance hierarchies, it is natural to expect that in many scenarios that hierarchy should be persistent, i.e that the *BeanContext* should participate in persistence mechanisms, in particular, either *java.io.serializable* or *java.io.externalizable*.

In particular *BeanContext*s shall persist and restore their current children that implement the appropriate persistence interfaces when they themselves are made persistent or subsequently restored.

As a result of the above requirement, persistent *BeanContextChil*d implementations are required to <u>not </u>persist any references to either their nesting *BeanContext*, or to any Delegates obtained via its nesting *BeanContext*.

BeanContexts shall, when restoring an instance of *BeanContextChild* from its persistence state, be required to invoke `setBeanContext()` on the newly instantiated *BeanContextChild,* with the actual parameter *beanContext* = to a reference to itself, the nesting *BeanContext*, in order to notify the newly restored instance of its nesting *BeanContext*, thus allowing that *BeanContextChild* to fully reestablish its dependencies on its environment.

Also note that since *BeanContext* implements *java.beans.BeanContextChild* it shall obey the persistence requirements defined below for implementors of that interface.

2.2 interface java.beans.BeanContextChild[1]

Simple JavaBeans that do not require any support or knowledge of their environment shall continue to function as they do today. However both JavaBeans that wish to utilize their containing *BeanContext*, and *BeanContext*s that may be nested, require to implement a mechanism that enables the propagation of the reference to the enclosing *BeanContext* through to cognizant JavaBeans and nested *BeanContext*s, the interface proposed is:

```
public interface java.beans.BeanContextChild
       extends   BeanContextListener {
     void        setBeanContext(BeanContext bc)
                    throws PropertyVetoException;

     BeanContext getBeanContext();

     void addPropertyChangeListener
          (String name, PropertyChangeListener pcl);
```

1. I don’t like this name much but I am struggling for a better alternative!

```
     void removePropertyChangeListener
          (String name, PropertyChangeListener pcl);

     void addVetoableChangeListener
          (String name, VetoableChangeListener pcl);

     void removeVetoableChangeListener
          (String name, VetoableChangeListener pcl);

}
```

When the *childStateChanged()* event is delivered to a JavaBean or *BeanContext*, the JavaBean or *BeanContext* may typically, as a side effect of this invocation, initialize, update, or invalidate any attributes or dependencies that it may have on its nesting *BeanContext's* environment, or as accessed via Delegation through that *BeanContext*.

A *BeanContextChild* object may throw a *ChildVetoException*, to notify the nesting *BeanContext* that it is unable to function/be nested within that particular *BeanContext*. Such a veto shall be interpreted by a *BeanContext* as an indication that the *BeanContextChild* has determined that it is unable to function in that particular *BeanContext* and is final.

During the unnesting of a *BeanContextChild* from its *BeanContext*, it is possible for the child to throw a *PropertyVetoException* to notify the caller that it is not in a state to be unnested. In order to bound this interaction a *BeanContextChild* may veto the initial unnesting notification, but may not veto any subsequent notifications, and must, upon receipt of such notifications, amend its state accordingly.

Note that classes that implement this interface, also act as an Event Source for (sub)interface(s) of *java.beans.PropertyChangeListener*, and are required to update their state accordingly and subsequently fire the appropriate *java.beans.PropertyChangeEvent* with *propertyName* = “beanContext”, *oldValue* = the reference to the previous nesting *BeanContext*, and *newValue* = the reference to the new nesting *BeanContext*, to notify any Listeners that its nesting *BeanContext* has changed value.

JavaBeans, or nested *BeanContext*s in the process of terminating themselves, shall invoke the *removeChildren()* method on their nesting *BeanContext* in order to withdraw themselves from the hierarchy prior to termination.

2.2.1 Important Persistence considerations

Instances of *BeanContextChild* nested within an *BeanContext*, will typically define fields or instance variables that will contain references to their nesting *BeanContext* instance, and possibly Delegates obtained from that *BeanContext* instance via its *getContextServices()* interface.

In order to ensure that the act of making such an instance persistent does not erroneously persist objects from the instances nesting environment, such instances shall be required to define such fields, or instance variables as `transient`.

This requirement is crucial since operations such as cutting and pasting object instances through a clipboard via object serialization will not function correctly if the act of serializing the target object also serializes the entire source environment it is nested within.

3.0 Overloading java.beans.instantiate() static method

Since *java.beans.instantiate()* is the current mechanism for (re)instantiating JavaBeans we need to extend or overload the syntax and semantics of this method in order to accommodate the introduction of the *BeanContext* abstraction. The extension proposed is:

```
public static Object instantiate(ClassLoader cl,
                                 String      beanName,
                                 BeanContext beanContext);
```

This method behaves has it is currently defined in the JavaBeans specification except in the case when the JavaBean instantiated implements the *java.beans.BeanContextChild* interface, in this case, the method invokes the *addChild()* method on the *beanContext* actual parameter with the value of the *targetChild* actual parameter = a reference to the newly instantiated JavaBean.[1]

4.0 Providing better support for Beans that are also Applets

The current implementation of *java.beans.instantiate()* contains minimal support for instantiating JavaBeans that are also Applets. In particular, this method will currently construct an *AppletContext* and *AppletStub* for the newly instantiated JavaBean, set the stub on the newly instantiated *Applet*, and *init()* the *Applet* if it has not already been invoked.

Unfortunately this does not provide sufficient support in order to allow most Applets to be fully functional, since the *AppletContext* and *AppletStub* created by *java.beans.instantiate()*, are noops. This is a direct consequence of the lack of sufficient specification of how to construct *AppletContext* and *AppletStub* implementations in the existing *Applet* API's. Furthermore, even if such specifications existed we would require an API that propagated a number of *Applet* attributes such as its **Codebase**, Parameters, *AppletContext*, and **Documentbase** into *java.beans.instantiate()* in order for it to subsequently instantiate the appropriately initialized objects.

1. Note: Since simple JavaBeans have no knowledge of a BeanContext, it is not advisable to introduce such instances into the hierarchy since there is no mechanism for these simple JavaBeans to remove themselves from the hierarchy and thus subsequently be garbage collected.

Since key to supporting fully functional Applets is to provide them with fully functional *AppletContext* and *AppletStub* instances, the design goal is to provide a mechanism to provide this state to *instantiate()* so that it may carry out the appropriate initialization and binding[1], therefore the proposed interface is:

```
public static Object
               instantiate(ClassLoader       cl,
                           String            beanName,
                           BeanContext       bCtxt,
                           AppletInitializer ai
               );
```

```
public interface AppletInitializer {
     void initialize(Applet newApplet, BeanContext bCtxt);
     void activate(Applet newApplet);
}
```

If the newly instantiated JavaBean is an instance of *java.applet.Apple*t then the new constructed *Applet*, (Bean) will be passed to the *AppletInitializer* via a call to *initialize().*

Compliant implementations of *AppletInitializer.initialize()* shall:

1. Associate the newly instantiated *Applet* with the appropriate *AppletContext*.
2. Instantiate an *AppletStub*() and associate that *AppletStub* with the *Applet* via an invocation of *setStub*().
3. If *BeanContext* parameter is `null`, then it shall associate the *Applet* with its appropriate *Container* by adding that *Applet* to its *Container* via an invocation of *add*(). If the *BeanContext* parameter is non-`null`, then it is the responsibility of the *BeanContext* to associate the *Applet* with its *Container* during the subsequent invocation of its *addChildren*() method.

Compliant implementations of *AppletInitializer.activate()* shall mark the *Applet* as active, and may optionally also invoke the *Applet*'s *start()* method.

Note that if the newly instantiated JavaBean is not an instance of *Applet*, then the *AppletInitializer* interface is ignored.

1. *AppletContext* objects expose a list of *Applet* objects they "contain", unfortunately the current *Applet* or *AppletStub* API's as defined, provide no mechanism for the *AppletContext* to discover its *Applets* from its *AppletStubs*, or for an *AppletStub* to inform its *AppletContext* of its *Applet.* Therefore we will have to assume that this binding/discovery can occur in order for this mechanism to be worthwhile in *java.beans.instantiate()*.

5.0 Standard/Suggested Conventions for BeanContext Delegates

5.0.1 BeanContexts that support InfoBus.

The InfoBus technology is a standard extension package that is intended to facilitate the rendezvous and exchange of dynamic self describing data, based upon a publish and subscribe abstraction, between JavaBean Components within a single Java Virtual Machine.

A *BeanContext* that that exposes an *InfoBus* to its nested *BeanContextChild* shall do so by exposing a service via the *hasService*() and *getService*() methods of type *javax.infobus.InfoBus.*

Thus *BeanContextChild* implementations may locate a common *InfoBus* implementation for their current *BeanContext* by using this mechanism to rendezvous with that *InfoBus* instance.

5.0.2 BeanContexts that support printing

A *BeanContext* that wishes to expose printing facilities to its descendants may delegate a reference of (sub)type *java.awt.PrintJob.*

5.0.3 BeanContext Design/Runtime mode support.

JavaBeans support the concepts of “design”-mode, when JavaBeans are being manipulated and composed by a developer in an Application Builder or IDE, and “Run”-mode, when the resulting JavaBeans are instantiated at runtime as part of an *Applet*, Application or some other executable abstraction.

In the first version of the specification, the “mode” or state, that is “design”-time or “run”-time was a JVM global attribute. This is insufficient since, for example, in an Application Builder environment, there may be JavaBeans that function, in “run”-mode, as part of the Application Builder environment itself, as well as the JavaBeans that function, in “design”-mode, under construction by the developer using the Application Builder to compose an application.

Therefore we require the ability to scope this “mode” at a granularity below that of JVM global.

The *BeanContext* abstraction, as a “Container” or “Context” for one or more JavaBeans provides appropriate mechanism to better scope this “mode”.

Thus *BeanContext*’s that wish to expose and propagate this “mode” to its descendants may delegate a reference of type *java.beans.BeanContextMode*:

```
public interface java.beans.DesignMode {
     void    setDesignTime(boolean isDesignTime);
     boolean isDesignTime();
}
```

Additionally, *BeanContext*s delegating such a reference shall be required to fire the appropriate *java.beans.propertyChangeEvent*, with propertyName = “designTime”, with the appropriate values for *oldValue* and *newValue,* when the “mode” changes value.

Note that it is illegal for instances of *BeanContextChild* to call *setDesignTime()* on instances of *BeanContext* that they are nested within.

5.0.4 BeanContext Visibility support.

JavaBeans with associated presentation, or GUI, may be instantiated in environments where the ability to present that GUI is either not physically possible (when the hardware is not present), or is not appropriate under the current conditions (running in a server context instead of a client).

The first version of the JavaBeans Specification introduced the *java.beans.Visibility* interface in order to provide a mechanism for JavaBeans to have their “visible” state, or ability to render a GUI, controlled from their environment.

*BeanContext*s that wish to enforce a particular policy regarding the ability of their children to present GUI, should use the *java.beans.Visibility* interface to control their children, however this mechanism in of itself may be insufficient.

Therefore *BeanContexts* that implement visibility policy shall delegate a reference of type *java.beans.visibilityState*[1]*:*

```
public interface java.beans.VisibilityState {
     boolean isOkToUseGui();
}
```

Additionally, *BeanContexts* delegating such a reference shall be required to fire the appropriate *java.beans.propertyChangeEvent*, with *propertyName* = “okToUseGui”, with the appropriate values for *oldValue* and *newValue,* when the “state” changes value.

Children of a *BeanContext* instance that does not delegate such an interface shall assume that it is permitted to render their associated GUI, if any, at any time.

The mechanism for setting the value of the “state”, is implementation dependent, but would typically be implemented or delegated through a *java.beans.visibility* interface.

5.0.5 Determining Locale from a BeanContext

*BeanContext*s may have a locale associated with them, in order to associate and propagate this important attribute across the JavaBeans nested therein.

1. Reusing java.beans.visibility here, instead of defining a new interface, would be nice but since it combines setters and getters it seems unsuitable as a mechanism for propagating state down the hierarchy since it would also (theoretically) allow it to propagate up also.

Therefore, *BeanContexts*, shall be required to fire the appropriate *java.beans.Property-ChangeEvent*, with propertyName = “locale”, *oldValue* = the reference to the previous value of the *Locale* delegate, and *newValue* = the reference to the new value of the *Locale* delegate, in order to notify its Listeners of any change in *Locale*.

Setting and getting the value of the *Locale* on the *BeanContext* is implementation dependent.

5.0.6 BeanContexts or JavaBeans that have associated presentation.

JavaBeans and *BeanContexts* that are associated with the presentation of a GUI shall either directly implement, or delegate a reference to, *java.awt.Component* and/or *java.awt.Container.*

During the invocation of *add()* the nesting *BeanContext* implementation may determine the *java.awt.Component* (if any) of the child it is adding and perform the necessary steps to cause its own *Container* and the child’s *Component* to be associated as defined by the *Container*’s *add*() semantics.

Similarly, during *remove()* the nesting *BeanContext* shall disassociate the child’s *Component* (if any) from its own *Container.*

6.0 java.beans.beancontext.BeanContextSupport

In order to ease the implementation of this relatively complex protocol a “helper” class is provided; *java.beans.beancontext.BeanContextSupport*. This class is designed to either be subclassed, or delegated (either explicitly or implicitly) by another object, and provides a fully compliant (extensible) implementation of the protocols embodied herein.